UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Dukič

Procesiranje slik s programskim jezikom Halide

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE
STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2017

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Dukič

Procesiranje slik s programskim jezikom Halide

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE
STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Danijel Skočaj

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge: Procesiranje slik s programskim jezikom Halide

Programiranje aplikacij za procesiranje slik je specifično, saj pogosto zahteva izračun novih vrednosti vseh slikovnih elementov. Zato je pri programiranju pogosta uporaba zank, ki poskrbijo za obdelavo vseh delov slike. Takšni problemi pa so zelo primerni za paralelizacijo. Pred kratkim se je pojavil namenski programski jezik z vgrajenimi mehanizmi, ki proces paralelizacije avtomatizirajo in nasploh poenostavijo pisanje programov za procesiranje slik. V diplomskem delu predstavite programski jezik Halide in implementirajte nekaj algoritmov za procesiranje slik. Izvajanje teh algoritmov primerjajte z izvajanjem algoritmov, implementiranih v programskem jeziku C++, in z algoritmi, implementiranimi v namenski knjižnici OpenCV.

Zahvaljujem se družini in prijateljem za njihovo podporo.

Kazalo

Seznam uporabljenih kratic

Kratica	Angleško	Slovensko
JIT	just in time	pravočasno
AOT	ahead of time	vnaprejšnje
SQDIFF	sum of squared difference	vsota razlik kvadratov
SQDIFF_NORM	normalized sum of squared difference	normalizirana vsota razlik kvadratov
CCORR	cross-correlation	navzkrižna korelacija
CCORR_NORM	normalized cross-correlation	normalizirana navzkrižna korelacija
CCOEFF	correlation coefficient	koeficient korelacije
CCOEFF_NORM	normalized correlation coefficient	normaliziran koeficient korelacije
ROI	region of interest	področje zanimanja

Povzetek

Naslov: Procesiranje slik s programskim jezikom Halide

V diplomskem delu je predstavljen novejši programski jezik za obdelavo slik Halide in njegova že uveljavljena alternativa OpenCV ter njuna primerjava. Ta obsega čas, potreben za izvedbo funkcij in reševanje enakih problemov, ter dolžino implementacij v številu vrstic. Problemi se stopnjujejo po kompleksnosti. Ti obsegajo implementacije za morfološke operacije in operacijo za zaznavanje objekta na sliki z uporabo vzorcev.

Vse operacije bodo opisane in implementirane na štiri načine. Prvi bo izdelan v jeziku C++ in bo za predstavitev slik uporabljal objekte iz OpenCV, drugi bo uporabljal funkcijo iz OpenCV, tretji in četrti pa bosta izdelana v programskem jeziku Halide. Razlika bo v tem, da eden ne bo uporabljal razvrščanja, zato bo bolj počasen.

V drugi polovici diplomskega dela so rezultati predstavljeni s tabelami, na koncu pa je še obrazložitev rezultatov.

Ključne besede: Halide, OpenCV, morfološke operacije, ujemanje vzorca, obdelava slik, računalniški vid

Abstract

Title: Processing images with programming language Halide

The thesis contains a presentation of a recently created programming language Halide and its comparison to an already established image processing library OpenCV. We compare the execution times of the implementations with the same functionality and their length (in terms of number of lines). The implementations consist of morphological operations and template matching.

Operations are implemented in four versions. The first version is made in C++ and only uses OpenCV's objects. The second version uses the OpenCV's function. The third and fourth version are written in Halide language. The difference between them is in the optimization. One of them uses scheduling while the other does not.

The second part of the thesis consists of the results which are shown in tables and the analysis of the results.

Keywords: Halide, OpenCV, morphology, template matching, image processing, machine vision

Poglavje 1 Uvod

Pri razvoju aplikacij za procesiranje slik se je treba odločiti, kateri programski jezik sploh želimo pri tem uporabiti. Zelo pogosto se odločitev zoži na jezike C++, Java ali Python, saj so vsi trije primerni. Recimo, da se odločimo za enega izmed teh in začnemo programirati. Začetek je sicer enostaven, vendar se z večjim obsegom berljivost kode hitro zmanjša in na koncu je zelo težko najti določen del kode, ki ga želimo spremeniti, saj večino kode obsegajo zanke, s katerimi določamo, kateri del slike želimo obdelovati. Recimo, da nam na koncu uspe izdelati program, ki ga potem poženemo, in pri tem ugotovimo, da je procesiranje zelo počasno, zato se odločimo kodo optimizirati. Logično je, da vestno testiramo program zato, da zagotovimo enak rezultat. Do popolne optimizacije je zato lahko potrebno zelo veliko časa. Čez čas se začnemo spraševati, ali obstaja kakšen bolj primeren jezik.

Programski jezik Halide je eden izmed takih jezikov, saj zaradi načina delovanja odstrani potrebo po zankah, s čimer zelo skrajša celotno kodo, pa vendar ohrani celotno funkcionalnost. Prav tako z ločitvijo definiranja operacije in njene optimizacije prepreči možnost, da bi se delovanje operacije spremenilo, kar pomeni, da je iskanje najboljše optimizacije zelo olajšano in posledično hitrejše.

V prvem delu diplomskega dela bo podrobneje opisano delovanja jezika Halide in knjižnice OpenCV, ki je trenutno ena izmed najbolj popularnih in posledično najbolj primerna za primerjavo delovanja. Nato sledi opis uporabljenih operacij, ki si med seboj razlikujejo po zahtevnosti. V drugem delu dela sledijo prikazi časov izvajanja raznih operacij in njihovo povprečje ter obrazložitev meritev.

Poglavje 2 Halide

2.1 Opis programskega jezika

Jezik Halide je leta 2012 ustvaril Jonathan Ragan-Kelley [5]. Gre za funkcijski jezik, kar pomeni, da je večina objektov in operacij predstavljena v obliki funkcij. Te so običajno definirane znotraj neskončne domene, kar pomeni, da niso odvisne od dimenzij vhodnih slik. V primeru izvajanja operacij, ki potrebujejo strukturni element, je mogoče definirati redukcijske funkcije, s katerimi se nato določi omejena domena (npr. definicija strukturnega elementa pri izvajanju morfoloških operacij). Zaradi tega so implementacije programov v obliki cevovodov, ki jih sestavljajo različne funkcije, ki opisujejo določen korak v operaciji.

Poleg tega omogoča enostavno optimiziranje programov z uporabo razvrščanja, ki je ločeno od implementacije operacij. Na tak način se lahko določi, kdaj se računa določene vrednosti in kam se jih shrani, kar je uporabno v primeru operacij, ki so razdeljene na več faz (npr. odpiranje) [5][8].

Prevajalnik jezika Halide na osnovi podane definicije in optimizacij zgenerira vektorizirano večnitno kodo, ki je odvisna od računalnika, na katerem se koda prevaja (v primeru, da se parametrov ne nastavi) [7]. Ta lahko prevede kodo na dva načina: na način JIT (angl. just in time) in na način AOT (angl. ahead of time). Pri prvem načinu se koda prevede, tik preden se jo požene, pri drugem pa se prevede v datoteko, ki jo je nato mogoče uporabiti tudi na računalnikih, ki nimajo naloženega jezika.

2.1.1 Implementacija funkcij

Halide je funkcijski jezik, kar pomeni, da je programiranje drugačno od objektno usmerjenega načina. Slike so predstavljene kot funkcije, ki kot argumente sprejmejo koordinate slike, vrnejo pa vrednost slikovnega elementa, nahajajočega se na teh koordinatah [5][6].

Potek procesiranja je opisan kot cevovod, sestavljen iz več funkcij, ki so lahko izrazi ali redukcijske funkcije. Izrazi so lahko [5][6]:

- aritmetične ali logične operacije,
- nalaganje slike,
- pogojni stavki,

- referenciranje vrednosti z imeni,
- klici drugih funkcij.

Redukcijske funkcije so namenjene implementiranju operacij, ki za delovanje potrebujejo podano domeno, npr. izvajanje konvolucije. Te so definirane rekurzivno in se ločujejo na funkcijo, ki določi začetno vrednost, in funkcijo, ki na podlagi določenega izraza ter predhodnih vrednosti poda rezultat izraza [6]. Na primer pri eroziji se na podlagi strukturnega elementa in vrednosti istoležnih slikovnih elementov na sliki določi vrednost centralnega slikovnega elementa. Za boljšo predstavo, kako se implementira redukcijske funkcije, je podan primer implementacije izdelave histograma v članku [6].

2.1.2 Opis razvrščanja

Za optimizacijo kode je treba uporabiti funkcije za razvrščanje, s katerimi lahko določamo, v kakšnem vrstnem redu se slikovni elementi obdelujejo in kateri rezultati se shranijo za nadaljnjo uporabo. S tem lahko vplivamo na hitrost izvajanja celotne operacije, saj lahko tako poskrbimo, da se slikovni elementi procesirajo v optimalnem vrstnem redu ter da se minimizira število redundantnih operacij. To je še posebej učinkovito v primerih, ko izvajamo operacije, sestavljene iz več korakov, kot npr. ujemanje vzorca ali morfološki gradient. Trenutno jezik podpira štiri načine razvrščanja.

Prvi način je vrivanje (angl. inline), pri katerem se slikovni element, ki ga potrebuje funkcija, obdela v vmesni fazi takrat, ko ga funkcija potrebuje. Redukcijske funkcije ni mogoče razvrščati v tem načinu, saj se funkcija izvede na celotni domeni. Ta način je lahko uporaben v primerih, ko ni veliko prostora za shranjevanje vmesnih rezultatov.

Drugi način je korenski način (angl. root). Pri tem načinu se pridobi vmesne rezultate za celotno sliko in šele nato se začne naslednja faza. Na ta način se vsak slikovni element obdela le enkrat, vendar pa je za to potrebnega veliko prostora.

Tretji način je paketni način (angl. chunk). Pri tem se razdeli sliko na dele in se nato pridobi vmesne rezultate samo za tisti del, ki se procesira. Ko se iteracija konča, se odstrani vmesne rezultate iz spomina. Za delitev se uporabi implementirane funkcije znotraj jezika.

Četrti način, imenovan ponovna uporaba (angl. reuse), je ta, da se rezultate, ki so shranjeni v spominu, uporabi v več funkcijah. Ta načina je možen le v primerih, ko se sliko procesira ali v korenskem načinu ali v paketnem načinu, in je najbolj primeren, ko izvajamo več funkcij, ki potrebujejo enake podatke.

2.2 Prevajanje kode

Izdelan program v jeziku Halide je mogoče prevesti na dva načina. Prvi način je način JIT (angl. just in time), ki je namenjen primerom, ki niso sestavni del večjih funkcij. Pri tem načinu se koda lahko ali prevede s posebno funkcijo *compile_jit* in se jo nato požene s funkcijo *realize*, lahko pa se požene samo funkcijo *realize*, ki bo pri prvem klicu prevedla funkcijo, in bo posledično počasnejša.

Drugi način je način AOT (angl. ahead of time), ki omogoča prevajanje kode v posebno datoteko. Ta način je primernejši, če je funkcijo treba klicati večkrat ali če želimo ustvariti funkcije v posameznih datotekah in jih nato uporabiti v ustvarjanju kompleksnejših funkcij (npr. implementacija odprtja z uporabo datotek, ki vsebujejo implementacijo erozije in širitve). Prav tako se lahko pri prevajanju v datoteko dodajo vse funkcije, potrebne za zagon, kar pomeni, da ni treba imeti knjižnice Halide na računalniku, na katerem se bo funkcija poganjala.

2.3 Uporabljeni objekti

Pri programiranju v jeziku Halide je treba uporabljati posebne objekte. Ti lahko predstavljajo vse od koordinat slikovnega elementa (angl. pixel) do celotnih slik. Prav tako so ločeni glede na to, kateri način prevajanja uporabljamo (način JIT ali način AOT). Sami smo uporabljali le del teh in so tudi opisani v članku [8]. Uporabljeni objekti so:

Var – običajno se uporablja za opisovanje značilnosti slikovnega elementa, in sicer koordinat ali barvnega kanala;

Func – predstavlja določeno stopnjo operacije;

Expr – vsebuje definicijo operacije ali le del te, uporablja se za določanje delovanja objekta tipa *Func*;

Image<X> – predstavlja sliko, ki je lahko tako vhod kot izhod določene funkcije. Prav tako je mogoče naložiti sliko in jo shraniti v objekt z uporabo funkcije *load_image* ali pa shraniti objekt z uporabo funkcije *save_image*. X predstavlja tip slikovnega elementa. Ti so lahko tipa:

- uint8_t, uint16_t, uint32_t, uint64_t,
- int8_t, int16_t, int32_t, int64_t,
- float ali
- double;

Tuple (z njim je mogoče vrniti več vrednosti različnih tipov kot rezultat določene funkcije. Rezultati bodo vedno v istem vrstnem redu.

V primeru, da se izdeluje koda *AOT*, je treba uporabljati naslednje objekte:

ImageParam (alternativa objektu *Image<X>*. Pri definiciji se določi tip podatkov z uporabo funkcije *type_of<X>()* ter število dimenzij. *X* predstavlja tip podatka.);

Param<X> (določa argument za klic prevedene kode. X predstavlja tip podatka.);

Target (objekt se uporablja za določanje lastnosti sistema, za katerega se koda prevaja. Z njim je mogoče določiti:

- vrsto operacijskega sistema,
- arhitekturo procesorja in
- dodatne funkcije).

2.4 Razvrščanje

Kot je že v opisu jezika omenjeno, je prednost jezika Halide enostavnost optimizacije algoritma. Jezik vsebuje osnovne funkcije, ki določajo vrstni red generiranih zank (npr. s tem se lahko določi, da se na določenih koordinatah najprej sprocesira vse barvne kanale in se zatem procesira naslednji slikovni element). Prav tako pa obstajajo tudi funkcije, ki predstavljajo nekatere kombinacije osnovnih funkcij. Ti sta npr. vektorizacija in način tiling.

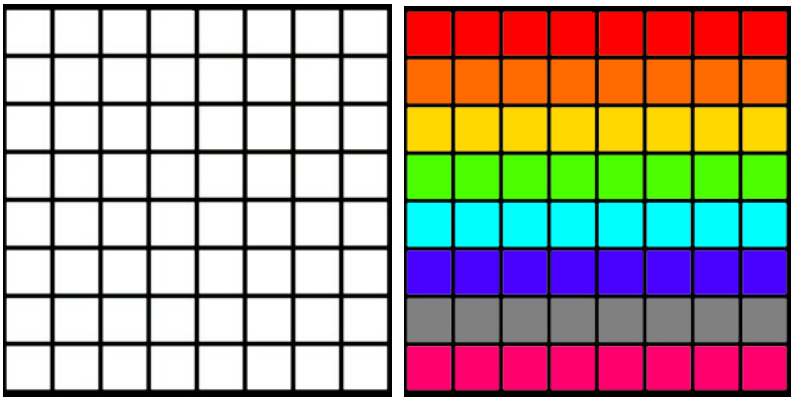
2.4.1 Osnovna razvrščanja

Te funkcije predstavljajo najbolj osnovne operacije, s katerimi nadzorujemo potek izvajanja operacije. Z njimi lahko spremenimo vrstni red zank (angl. reorder), razdelimo spremenljivko na več spremenljivk (angl. split) in obratno (angl. fuse). Posamezne funkcije na čas izvajanja nimajo velikega vpliva, nasprotno pa velja za kombinacije, ki lahko proces pospešijo ali pa tudi ne.

2.4.2 Vektorizacija

Vektorizacija (angl. vectorize) razdeli sliko na vektorje, na katerih se nato izvaja operacija. Prednost tega je izvajanje operacije na več slikovnih elementih hkrati in posledično povečanje hitrosti, tako kot je prikazano na sliki 2.4.2.1. V primeru, da dimenzija, ki se deli na vektorje, ni deljiva z velikostjo vektorja, se za izvedbo robnega primera (zadnjega vektorja) uporabijo

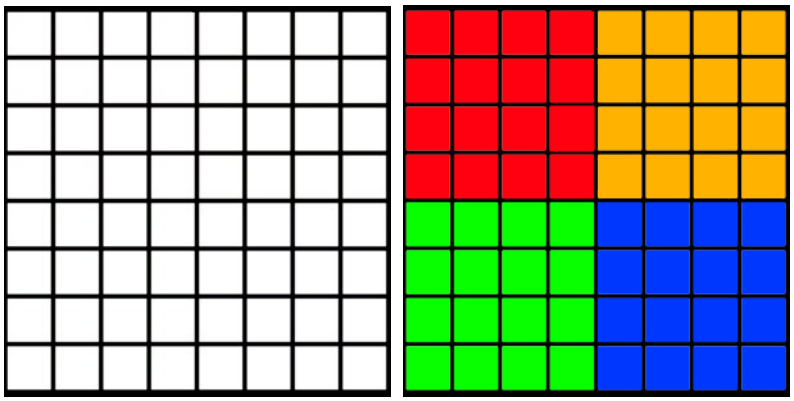
slikovni elementi iz prejšnjega vektorja, kar pomeni, da se za nekatere dele izvaja redundantno delo.



Slika 2.4.2.1: Združevanje slikovnih elementov v vektorje dolžine 8

2.4.3 Način tiling

Način tiling razdeli sliko na manjše dele, ki jih je nato mogoče izvajati paralelno. Primer načina je prikazan na sliki 2.4.3.1.



Slika 2.4.3.1: Združevanje slikovnih elementov v matrike velikosti 4 × 4 z uporabo načina tiling

2.5 Primeri uporabe

2.5.1 Primer kode JIT

Na sliki 2.5.1.1 je prikazan primer implementirane funkcije, ki se prevede na način JIT.

```
1  int main(int argc, char **argv) {
2
3   Image<uint8_t> input = load_image("images/rgb.png");
4
5   Func brighter;
6   Var x, y, c;
7
8   Expr value = input(x, y, c);
9
10  value = cast<float>(value);
11  value = value * 1.5f;
12  value = min(value, 255.0f);
13
14  value = cast<uint8_t>(value);
15
16  brighter(x, y, c) = value;
17  Image<uint8_t> output = brighter.realize(input.width(), input.height(),
input.channels());
18
19  save_image(output, "brighter.png");
20
21  return 0;
22 }
```

Slika 2.5.1.1: Funkcija za osvetlitev slike v načinu JIT

Koda se začne z nalaganjem slike, ki bo predmet obdelave (vrstica 3). Slikovni elementi naložene slike bodo tipa *uint8_t*, kar pomeni, da bodo imeli vrednosti med 0 in 255. Nato se deklarirajo objekt tipa *Func* **brighter** (vrstica 5) in trije objekti tipa *Var* **x**, **y** ter **c** (vrstica 6). **x** in **y** bosta vsebovala koordinate obdelovanega slikovnega elementa, **c** pa bo določal barvni kanal (R, G ali B). Glede na to, da se slika osvetljuje tako, da se trenutno vrednost slikovnega elementa množi s številko tipa *float*, se pred tem podatke slike pretvori v tip *float* (vrstica 10). Nato se izvede množenje (vrstica 11), zatem pa se poskrbi, da so podatki še vedno manjši od zgornje meje (255), to pa zato, da se pri pretvarjanju nazaj v tip *uint8_t* (vrstica 14) ne izgubijo rezultati (vrstica 12). Potem se celoten postopek shrani v objekt **brighter** (vrstica 16), s katerim nato tudi izvedemo operacijo (vrstica 17). Rezultat se shrani v objekt **output**, ki ga tudi shranimo v datoteko, imenovano **brighter.png** (vrstica 19).

2.5.2 Primer kode AOT

Na sliki 2.5.2.1 je prikazana koda funkcije v načinu AOT. Na sliki 2.5.2.2 pa je prikazan zagon prevedene funkcije.

```
1  int main(){
2
3   ImageParam input(Halide::type_of<uint8_t>(), 3);
4
5   Func brighter;
6   Var x, y, c;
7
8   Expr value = input(x, y, c);
9
10  value = cast<float>(value);
11  value = value * 1.5f;
12  value = min(value, 255.0f);
13
14  value = cast<uint8_t>(value);
15
16  brighter(x, y, c) = value;
17  brighter.compile_to_static_library("brighter", {input});
18
19  return 0;
20 }
```

Slika 2.5.2.1: Koda AOT za osvetlitev slike (prevajanje funkcije)

```
1  int main(){
2
3   string path;
4   string resultPath;
5
6   Image<uint8_t> input = load_image(path);
7   Image<uint8_t> output(input.width(), input.height(), input.channels());
8
9   brighter(input.raw_buffer(), output.raw_buffer());
10
11  save_image(output, resultPath);
12  return 0;
13 }
```

Slika 2.5.2.2: Koda AOT za osvetlitev slike (izvajanje funkcije)

Kot je razvidno, je koda, ki definira operacijo, enaka pri obeh načinih (AOT – 2.5.2.1, JIT – 2.5.1.1), razlike pa se pojavijo pri prevajanju in poganjanju funkcije. Prva razlika med obema načinoma je, da se pri načinu AOT koda loči na prevajanje in izvajanje. Prav tako se pri prevajanju uporablja drugačen objekt, ki predstavlja sliko. To je objekt tipa *ImageParam*, ki se uporablja kot parameter slike, ki bo prejela sliko kot argument pri izvedbi. Druga razlika je ta, da se pri prevajanju na koncu kliče funkcija *compile_to_static_library*, ki prevede operacijo v datoteko. Nato se datoteko uporabi v drugem projektu, prikazanem na sliki 2.5.2.2, kjer je postopek bolj podoben načinu JIT, vendar ima tudi ta del razlike pri izvajanju funkcije, saj za

izvedbo potrebuje objekte tipa *buffer_t*, ki je matrika slikovnih elementov in ne vsebuje podatkov o sliki [8].

Poglavje 3 OpenCV

OpenCV (angl. Open Source Computer Vision Library) je odprtokodna knjižnica, ki vsebuje več kot 2500 optimiziranih algoritmov za računalniški vid in strojno učenje [9]. Ti obsegajo vse od prepoznave objektov in ljudi (prepoznava obrazov na sliki, prepoznava ljudi na video posnetkih na podlagi primerov človeškega obnašanja itd.) pa do pridobivanja tridimenzionalnih modelov objektov na podlagi slik, sledenja premikom oči itd. OpenCV uporablja veliko znanih podjetij, npr. Google, Microsoft, Intel, IBM, kot tudi manjši start-upi (Applied Minds, VideoSurf itd.) [1] [9].

Knjižnica je izdelana za več jezikov. Ti so C++, C, Python, Java in MATLAB. Poleg tega podpira več operacijskih sistemov (Windows, Linux, Android, Mac OS). Prenešena je bila več kot 14 milijonkrat [11].

3.1 Primer kode C++ z uporabo objektov iz OpenCV

Na sliki 3.1.1 je prikazana implementacija funkcije, ki osvetli sliko v jeziku C++ brez uporabe funkcij iz knjižnice OpenCV.

```
1  int main()
2  {
3   string path = "";
4   string resultPath = "";
5   float value = 1.5f;
6
7   Mat input = imread(path);
8   Mat result = Mat::zeros(input.size(), input.type());
9
10  for (int y = 0; y < input.rows; y++){
11   for (int x = 0; x < input.cols; x++){
12    for (int c = 0; c < 3; c++){
13     result.at<Vec3b>(y, x)[c] = saturate_cast<uchar>(value *
(input.at<Vec3b>(y, x)[c]));
14    }
15   }
16  }
17
18  imwrite(resultPath, result);
19  return 0;
20 }
```

Slika 3.1.1: Koda C++ z uporabo objektov iz OpenCV

V vrsticah 3 in 4 se določi pot do slike, ki se jo spreminja, ter pot, kamor se spremenjena slika shrani. Objekt v vrstici 5 določa vrednost, s katero množimo slikovni element.

V vrstici 7 se definira objekt tipa *Mat*, v katerega se naloži slika. Za shranjevanje novih vrednosti slikovnih elementov se definira objekt **result**, ki je matrika enakih dimenzij kot slika in je zapolnjena z ničlami.

Vrstice 10–16 predstavljajo operacijo. Prvi dve zanki (vrstici 10 in 11) določata koordinate slikovnega elementa, tretja (vrstica 12) pa določa barvni kanal.

V vrstici 13 se izvaja osvetljevanje. S funkcijo *at* se pridobi vrednost slikovnega elementa na koordinatah **y**, **x**. Glede na to, da je slika barvna (ima 3 barvne kanale), in je posledično treba za vsak kanal izračunati novo vrednost, se uporablja razred *Vec3b*. To je vektor, ki vsebuje 3 elemente (vrednosti za barvne kanale R, G in B). Ti so tipa *uchar*, kar pomeni, da so vrednosti lahko 0–255. Funkcija *saturate_cast* se uporablja zato, da se zagotovi natančno pretvarjanje tipa podatkov [10]. To pomeni, da bo vrednosti, ki so izven meja tipa, v katere pretvarjamo, dodelila mejno vrednost.

Na koncu se rezultat shrani z uporabo funkcije *imwrite* v datoteko na dodeljeni poti (vrstica 18).

3.2 Primer kode C++ z uporabo funkcije OpenCV

Na sliki 3.2.1 je prikazan primer kode, ki osvetli sliko, izdelano v jeziku C++ z uporabo funkcije OpenCV *convertTo*.

V vrstici 3 se z uporabo funkcije *imread* naloži slika. Nato se deklarira objekt tipa *Mat* **output**, v katerega se bo shranila osvetljena slika.

V vrstici 6 se kliče funkcija *convertTo*, ki za delovanje potrebuje več argumentov. Prvi argument je objekt, v katerega se shrani rezultat funkcije, drugi argument določa, kakšen tip podatkov bo vsebovala slika (v primeru negativnega števila se uporabi tip podatkov iz vhodne slike). Tretji argument določa velikost slike (v primeru, da se uporabi vrednost 1, se ohranijo dimenzije slike). Četrti argument se prišteje vsem slikovnim elementom, kar posledično spremeni svetilnost slike.

Na koncu (v vrstici 7) se shrani predelana slika z uporabo funkcije *imwrite*.

```
1 int main()
2 {
3   Mat input = imread("pot\do\slike");
4   Mat output;
5
6   input.convertTo(output, -1, 1.0, 100.0);
7   imwrite("pot\do\shranjene\slike", output);
8   return 0;
9 }
```

Slika 3.2.1: Primer kode C++ z uporabo funkcije OpenCV

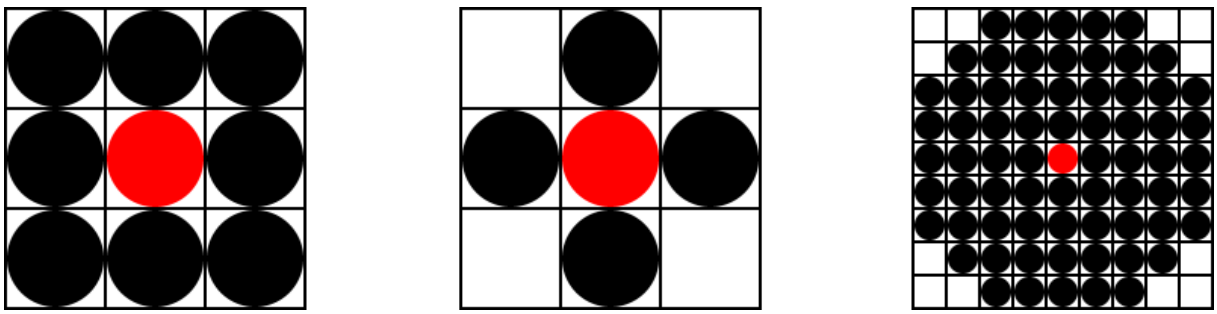
Poglavje 4 Implementirane operacije

4.1 Morfološke operacije

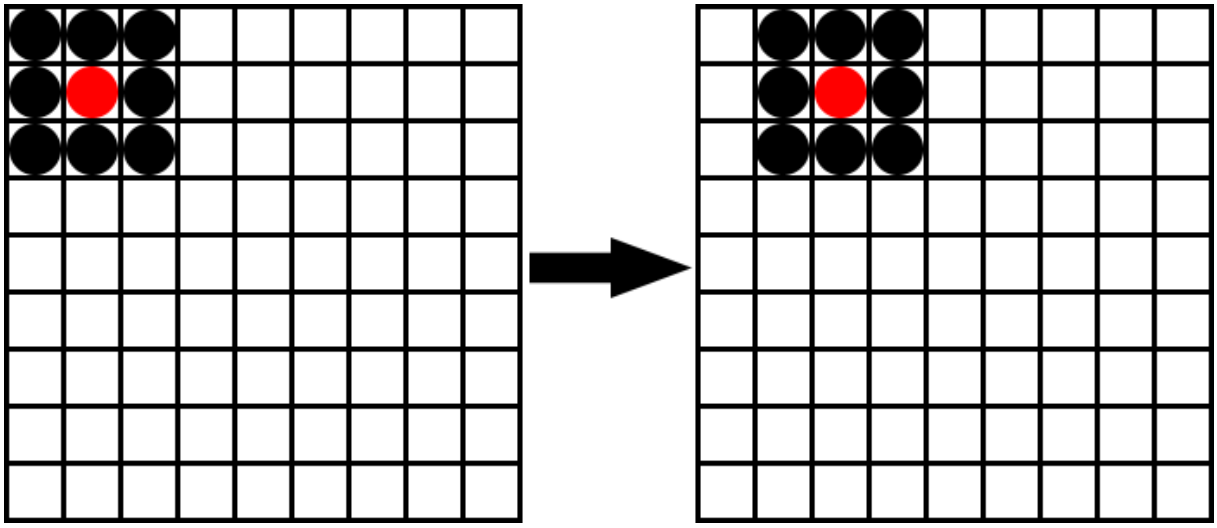
Morfološke operacije (angl. morphology) so zbirka operacij za obdelovanje slik. Za delovanje potrebuje sliko, na kateri se izvaja operacija, in strukturni element, ki določa, kateri podatki se uporabijo pri izvajanju operacije. Ta je lahko različnih oblik (npr. OpenCV ponuja kvadrat, križ in elipso, pri Halide pa se uporablja redukcijsko domeno za določanje oblike). Primeri strukturnih elementov so prikazani na sliki 4.1.1 pri katerih črni krogi predstavljajo slikovne elemente, zajete znotraj strukturnega elementa. Rdeči krogi predstavljajo slikovni element, v katerega se shrani rezultat operacije.

Prav tako je treba imeti binarno sliko, kar pomeni, da je sestavljena iz črnih in belih slikovnih elementov, ki se nato ločijo na ospredje in ozadje (v diplomskem delu so beli slikovni elementi označeni kot ospredje, črni pa kot ozadje). Operacije se izvajajo nad ospredjem.

Najbolj osnovni morfološki operaciji sta erozija in širitev. Operacije se začnejo tako, da se strukturni element postavi na začetek slike (tako da je centralni slikovni element postavljen na zgornji levi rob slike). Potem se izvede operacija, pri kateri se uporabijo podatki, zajeti v strukturnem elementu. Rezultat operacije se shrani v centralni slikovni element (v drugi matriki). Po izvajanju se strukturni element pomakne na naslednji slikovni element, vse dokler ne pride do zadnjega (primer na sliki 4.1.2).



Slika 4.1.1: Primeri strukturnega elementa s centralnim slikovnim elementom. Prvi prikazuje kvadrat, drugi križ, tretji pa krog



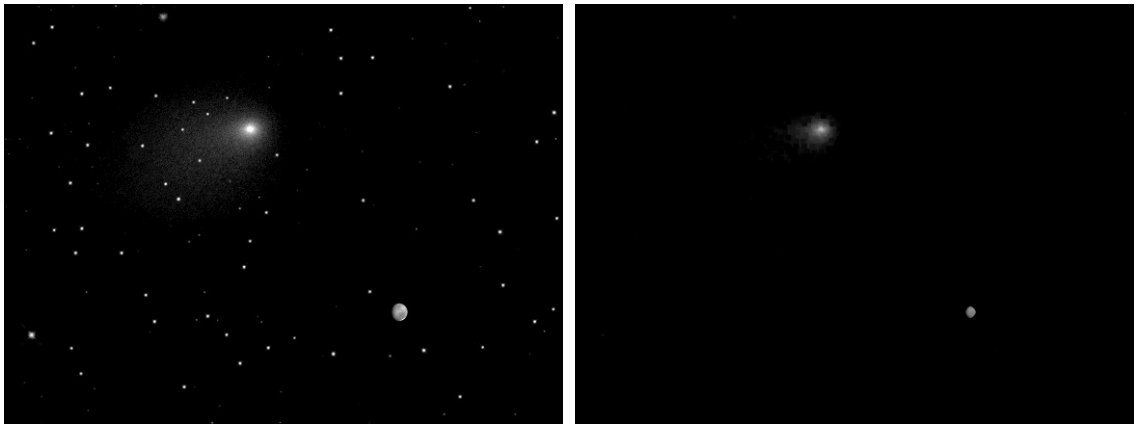
Slika 4.1.2: Primer premika strukturnega elementa

4.1.1 Erozija

Erozija (angl. erosion) je ena izmed dveh najbolj osnovnih morfoloških operacij, na katerih temeljijo zahtevnejše funkcije. Uporablja se lahko v primerih, ko želimo ožiti določene predmete (npr. slika 4.1.1.1) ali ko želimo odstraniti manjše objekte iz slike (npr. slika 4.1.1.2). Izvajanje operacije deluje tako, da pomikamo strukturni element (B) čez sliko (A). Dimenzija strukturnega elementa določa intenzivnost operacije. V primeru, da se v strukturnem elementu pojavi slikovni element (v enačbi označen s p), ki predstavlja ozadje, se centralni slikovni element (v enačbi označen z B_p) prebarva v barvo ozadja. Erozija je definirana z enačbo (4.1.1.1)[3].

$$A \ominus B = \{p : B_p \subset A\} \tag{4.1.1.1}$$

Slika 4.1.1.1: Ožanje znaka z uporabo erozije



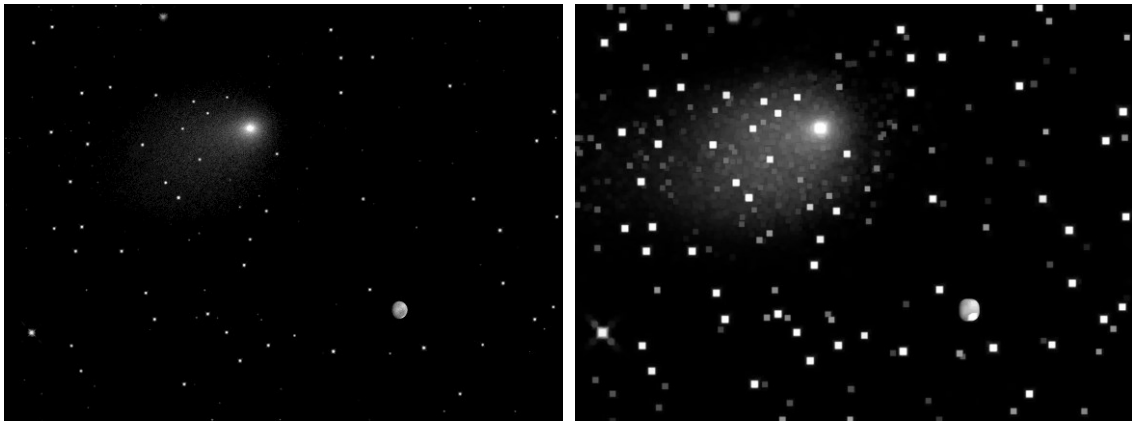
Slika 4.1.1.2: Odstranjevanje delcev iz slike z uporabo erozije

4.1.2 Širitev

Širitev (angl. dilation) je druga izmed najbolj osnovnih morfoloških operacij. Uporablja se v primerih, ko želimo povečati intenzivnost robov, kot na sliki 4.1.2.1 ali, ko želimo poudariti zvezde na sliki 4.1.2.2. Tako kot erozija tudi širitev potrebuje binarno sliko (A). Postopek poteka tako, da se preveri, ali se v strukturnem elementu (B) nahaja slikovni element, ki predstavlja ospredje (v enačbi označeno s črko p), in v primeru, da se, se potem obarva centralni slikovni element (v enačbi označen kot B_p) v barvo, ki predstavlja ospredje. Širitev opisuje enačba (4.1.2.1)[3].

$$A \oplus B = \{p : B_p \cap A \neq \emptyset\} \tag{4.1.2.1}$$

Slika 4.1.2.1: Poudarjanje znaka z uporabo širitve

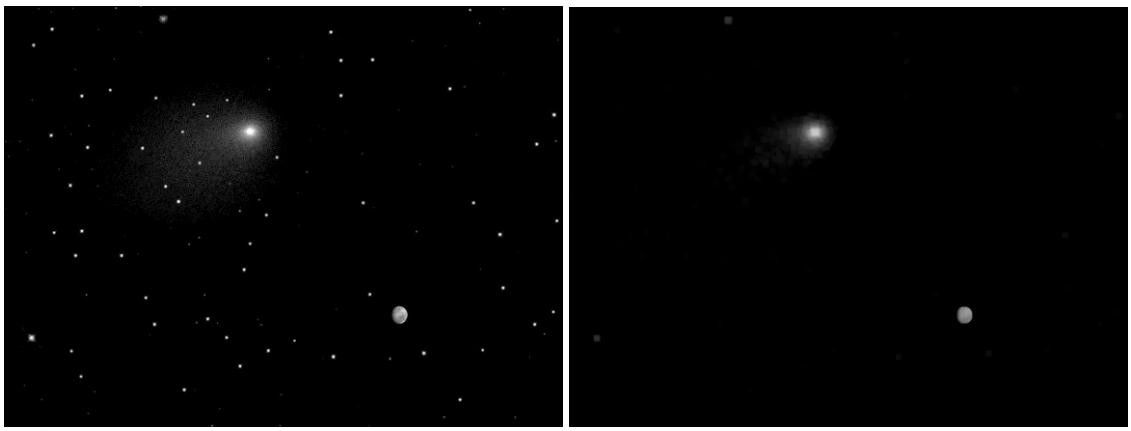


Slika 4.1.2.2: Povečevanje šuma v sliki z uporabo širitve

4.1.3 Odpiranje

Odpiranje (angl. opening) je rezultat širitve (v enačbi označeno kot $\oplus$) nad erozijo (v enačbi označeno kot $\ominus$) slike A z uporabo strukturnega elementa B. Za razliko od erozije in širitve se velikosti večine regij ospredja (v primeru, da ni odstranjena) ohranijo. Uporablja se za odstranitev delcev iz slike in posledično izboljšuje sliko za nadaljnjo obdelavo. Odpiranje opisuje enačba (4.1.3.1)[3]. Na sliki 4.1.3.1 je prikazan primer uporabe odpiranja.

$$A \circ B = (A \ominus B) \oplus B \tag{4.1.3.1}$$



Slika 4.1.3.1: Uporaba odpiranja nad sliko na levi strani

4.1.4 Zapiranje

Zapiranje (angl. closing) je rezultat erozije nad širitvijo slike *A* z uporabo strukturnega elementa *B*. Glede na to, da je delovanje zapiranja nasprotno od odpiranja, je uporabno za zapolnjevanje lukenj na slikah. Tako kot pri odpiranju se tudi pri zapiranju ohranjajo velikosti regij ospredja. Zapiranje opisuje enačba (4.1.4.1)[3]. Na sliki 4.1.4.1 je prikazan primer uporabe zapiranja.

$$A ¥ B = (A \oplus B) \ominus B \tag{4.1.4.1}$$

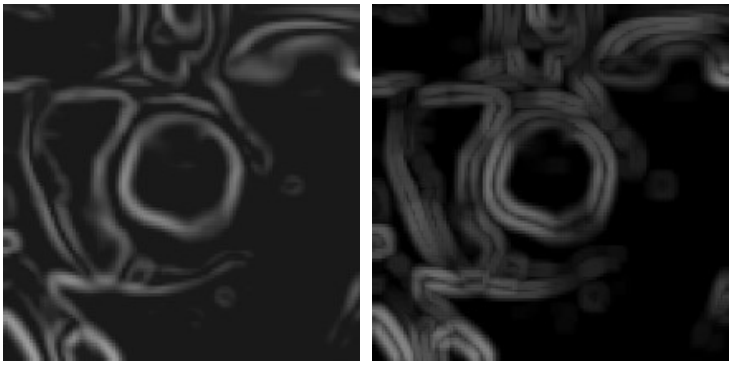
Slika 4.1.4.1: Uporaba zapiranja nad levim znakom s strukturnim elementom 5 × 5

4.1.5 Morfološki gradient

Morfološki gradient je razlika širitve in erozije ter je eden izmed načinov za iskanje robov objektov na obdelani sliki. Primera izvajanja morfološkega gradienta sta prikazana na slikah 4.1.5.1 in 4.1.5.2. Enačba za morfološki gradient je (4.1.5.1).

$$G(A) = A \oplus B - A \ominus B \tag{4.1.5.1}$$

Slika 4.1.5.1: Pridobivanje robov znaka z uporabo morfološkega gradienta



Slika 4.1.5.2: Pridobivanje robov celic na levi sliki z uporabo morfološkega gradienta

4.2 Ujemanje vzorca

Ujemanje vzorca (angl. template matching) je eden izmed načinov iskanja predmetov na sliki. Za učinkovito iskanje je treba imeti več vzorcev iskanega objekta, ki jih nato primerjamo z deli izvorne slike. Poznamo več različnih metod za pridobivanje rezultatov, ki vplivajo na hitrost in natančnost iskanja. Te so opisane v naslednjih enačbah, pri katerih R predstavlja matriko, v katero shranjujemo pridobljene vrednosti, T predstavlja vzorec in I predstavlja sliko. Meje seštevanja $(x',\ y')$ predstavljata dimenziji vzorca. x in y predstavljata pozicijo zgornjega levega kota dela slike, ki se trenutno uporablja. V te koordinate se prav tako shrani pridobljena vrednost [4]. Enačbe so:

- **SQDIFF (angl. sum of squared difference)** in normalizirana različica **SQDIFF_NORM** (normalized sum of squared difference):

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2 \tag{4.2.1}$$

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 * \sum_{x',y'} I(x+\ x',y+y')}} \tag{4.2.2}$$

- **CCOEFF (angl. correlation coefficient) in normalizirana različica CCOEFF_NORM** (angl. normalized correlation coefficient) (w predstavlja širino slike, h pa višino):

$$R(x,y) = \sum_{x',y'}(T'(x',y') * I(x+x',y+y')) \text{ , kjer} \tag{4.2.3}$$

$$T'(x',y') = T(x',y') - \frac{1}{w*h} * \sum_{x'',y''} T(x'',y'')$$

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{1}{w*h} * \sum_{x'',y''} I(x+x'',y+y'')$$

$$R(x,y) = \frac{R(x,y) = \sum_{x',y'}(T'(x',y') * I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 * \sum_{x',y'} I'(x+x',y+y')^2}} \tag{4.2.4}$$

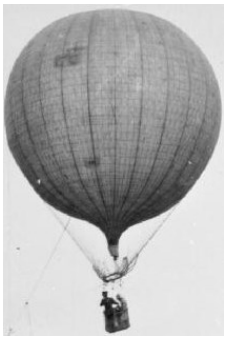
- **CCORR (angl. cross-correlation)** in normalizirana različica **CCORR_NORM** (angl. normalized cross-correlation):

$$R(x,y) = \sum_{x',y'} T(x',y') * I(x+x',y+y') \tag{4.2.5}$$

$$R(x,y) = \frac{\sum_{x',y'} T(x',y') * I(x+x',y+y')}{\sqrt{\sum_{x',y'} T(x',y')^2 * \sum_{x',y'} I(x+x',y+y')}} \tag{4.2.6}$$

Če se uporabljata SQDIFF ali SQDIFF_NORM, je treba biti pozoren na dejstvo, da nižji rezultat predstavlja boljše ujemanje, pri ostalih metodah pa je ravno obratno. Postopek pridobivanja vseh vrednosti za ugotovitev lokacije najboljšega ujemanja poteka tako, da se prekrije del slike z vzorcem (začne se na zgornjem levem robu in se pomika proti koncu slike), potem pa se uporabi ena izmed formul za izračun vrednosti na podlagi vseh slikovnih elementov v vzorcu ter v pokritem delu slike. Rezultat izračuna se nato shrani v posebno matriko. Ko se vse vrednosti pridobijo, se poišče najboljši rezultat, ki predstavlja lokacijo iskanega predmeta na sliki. V primeru, da želimo poiskati več predmetov, se doda mejno vrednost, ki določa spodnjo (ali zgornjo) mejo vseh vrednosti, ki jih upoštevamo kot ujemanje.

Kot primer je prikazan najden predmet na sliki 4.2.2 z uporabo vzorca, prikazanega na sliki 4.2.1.



Slika 4.2.1: Iskani predmet

Slika 4.2.2: Rezultat iskanja

Poglavje 5 Implementacija operacij

Operacije so implementirane na tri načine. Prvi način je implementacija v jeziku Halide. Ta uporablja prevajanje v načinu AOT in je zaradi tega koda ločena na implementacijo operacije in njen prevod v ločeno datoteko in zagon operacije (v ločenem programu). Glede na to, da je večina kode pri morfoloških operacijah enaka, je samo pri implementaciji erozije prikazana celotna koda. Pri ostalih implementacijah so prikazani samo deli kode, ki se ločuje od ostalih.

Drugi način je implementacija z uporabo funkcije OpenCV. Pri tem se uporablja temu namenjena funkcija, imenovana *morphologyEx,* ki zajema vse morfološke operacije. Zaradi tega je prikazana samo implementacija erozije.

Tretji način je implementacija v C++ z uporabo objektov iz OpenCV, ne pa funkcij. Pri tem načinu je prikazana implementacija erozije in morfološkega gradienta, ne pa širitve ter odprtja. To je zato, ker se pri odprtju kličeta implementirani funkciji erozije in širitve, pri širitvi pa je implementacija skoraj enaka eroziji.

5.1 Morfološke operacije

5.1.1 Erozija

Halide – prevajanje:

Na začetku implementacije, prikazane na sliki 5.1.1.1 (vrstice 2–5) so definirani objekti, ki bodo uporabljeni pri implementaciji algoritma. Objekt v vrstici 4 vsebuje podatek o dimenzijah strukturnega elementa (ta se določi ob klicu funkcije). Objekt v vrstici 5 je slika, ki bo podana ob klicu funkcije.

Glede na to, da obstaja možnost, da se bo strukturni element pomaknil izven meja slike, se v vrstici 7 določi, kateri podatki naj se uporabijo v takem primeru (v prikazanem primeru se uporabi podatke na robu slike) in se shranijo v objekt **limit**, ki se ga bo potem uporabljalo kot argument za izvedbo funkcije namesto slike.

Strukturni element je definiran v vrstici 9. Ta bo vedno kvadrat z dimenzijo **dimension**. V vrstici 11 je opisano delovanje algoritma (enačba (4.1.1.1)), v vrstici 13 pa optimizacija le-tega. Na koncu (vrstice 15–18) sledi še pridobitev podatkov o računalniku (operacijski sistem in arhitektura procesorja) ter klic funkcije za prevajanje kode v datoteko **erosion**.

```
1 void createAOTE() {
2   Var x, y;
3   Func erosion, limit;
4   Param<int> dimension;
5   ImageParam input(type_of<uint8_t>(), 2);
6
7   limit = BoundaryConditions::repeat_edge(input);
8
9   RDom r(-1 * dimension / 2, dimension, -1 * dimension / 2, dimension);
10
11  erosion(x, y) = argmin(r, limit(x + r.x, y + r.y))[2];
12
13  erosion.vectorize(x, 4).parallel(y);
14
15  Target t = get_host_target();
16  t.set_feature(Target::NoRuntime, true);
17  erosion.compile_to_static_library("erosion", { input, dimension }, t);
18 }
```

Slika 5.1.1.1: Koda za prevajanje funkcije za izvajanje erozije

Halide – zagon:

Celoten postopek, prikazan na sliki 5.1.1.2, se nahaja v funkciji **runE**, ki za pogon potrebuje lokacijo vhodne slike (**path**), dimenzijo strukturnega elementa (**dim**) in pot, kamor se shrani spremenjena slika (**out**). Glede na to, da se bo uporabljala slika kot vhodni argument, se bo uporabljalo objekt *Image*, v katerem bodo podatki tipa *float*. V vrsticah 2 in 3 se deklarirata objekta za vhodno ter izhodno sliko. Izhodna slika ima za argumente podane dimenzije vhodne slike. V vrstici 4 se izvede klic funkcije za erozijo. Kot argumente se poda kazalec na vsebino slike, dimenzijo strukturnega elementa (intenzivnost erozije je premo sorazmerna z dimenzijo strukturnega elementa), shranjeno v objektu **dim,** in kazalec na izhodno sliko. V vrstici 5 se shrani slika v datoteko, ki se nahaja na poti, shranjeni v objektu **out**.

Kljub temu da se izvaja koda AOT, je vseeno treba imeti knjižnico *Halide* in dodatek *Halide::Tools* za shranjevanje/nalaganje slik.

```
1 void runE(const std::string path, const int dim, const std::string out) {
2     Image<float> input = Halide::Tools::load_image(path);
3     Image<float> output(input.width(), input.height());
4     erosion(input.raw_buffer(), dim, output.raw_buffer());
5     Halide::Tools::save_image(output, out);
6 }
```

Slika 5.1.1.2: Koda za izvajanje prevedene funkcije za erozijo

Funkcija OpenCV

V implementaciji, prikazani na sliki 5.1.1.3, je mogoče uporabiti različne oblike strukturnih elementov (matrika, križ in elipsa). Funkcija *erode* je že implementirana znotraj OpenCV-ja.

Kot argumente potrebuje izvorno sliko, objekt, v katerega shranimo rezultat, in strukturni element.

```
1  void Erosion(int, void*) {
2   int erosion_type, int erosion_elem = 0;
3   if (erosion_elem == 0) { erosion_type = MORPH_RECT; }
4   else if (erosion_elem == 1) { erosion_type = MORPH_CROSS; }
5   else if (erosion_elem == 2) { erosion_type = MORPH_ELLIPSE; }
6
7   Mat element = getStructuringElement(erosion_type, Size(3,3),
8                       Point(erosion_size, erosion_size));
9
10  erode(srcEroDil, erosion_dst, element);
11 }
```

Slika 5.1.1.3: Koda za izvajanje erozije v jeziku z uporabo funkcije OpenCV

C++

Za delovanje implementacij, prikazane na sliki 5.1.1.4, potrebuje funkcija vhodno sliko (ki mora biti binarna) kot argument **input**. V vrstici 2 se definira objekt **output** z enakimi dimenzijami kot vhodna slika. **newValue** vsebuje novo vrednost centralnega slikovnega elementa v trenutnem strukturnem elementu.

Delovanje erozije je opisano v vrsticah 4–25. Prvi dve zanki (vrstici 4 in 5) določata koordinate centralnega slikovnega elementa in posledično premikata strukturni element. Drugi dve zanki (vrstici 6 in 7) premikata kazalec na trenutno obravnavani element v strukturnem elementu.

Pogojni stavek v vrstici 8 preverja, ali se obravnavani element nahaja izven meja slike, drugi pogojni stavek pa preverja, ali je element manjši od centralnega elementa, in shrani novo vrednost v objekt **newValue**. Posledično se tudi preverja, ali je centralni slikovni element bele barve. Centralnemu slikovnemu elementu se nato dodeli primerna vrednost v vrstici 21. V vrstici 24 se vrača objekt **output**, v katerem je shranjena spremenjena slika

```
1       Mat erosion(Mat input) {
2       Mat output(input.rows, input.cols, input.type());
3       uint8_t newValue = 255;
4       for (int x = 0; x < input.cols; x++){
5        for (int y = 0; y < input.rows; y++){
6         for (int xk = -1; xk < 3; xk++){ //kernel
7         for (int yk = -1; yk < 3; yk++) {
8          if ((x + xk) > -1 && (y + yk) > -1 && (x + xk) < input.cols &&
(y + yk) < input.rows) {
9           if (input.at<uint8_t>(Point(x + xk, y + yk)) < in-
put.at<uint8_t>(Point(x, y))) {
10           newValue = input.at<uint8_t>(Point(x + xk, y + yk));
11          }
12          else {
13           newValue = 255;
14          }
15         }
16         else {
17          newValue = 0;
18         }
19        }
20       }
21       output.at<uint8_t>(Point(x, y)) = newValue;
22      }
23     }
24     return output;
25    }
```

Slika 5.1.1.4: Koda za izvajanje erozije v jeziku C++

5.1.2 Širitev

Halide – prevajanje:

V implementaciji, prikazani na sliki 5.1.2.1, se v vrstici 1 se deklarira objekt tipa *Func* **dilate**. Nato se mu dodeli delovanje v vrstici 3. Pri tem se določi, da se funkcija izvede pri vsakem slikovnem elementu na koordinatah **x** in **y**. Funkcija vrne objekt tipa *Tuple*, ki vsebuje najvišjo vrednost in koordinate slikovnega elementa znotraj redukcijske domene (v tem primeru določeno z objektom **r**). Glede na to, da potrebujemo samo vrednost in ne koordinat, je na koncu to tudi določeno (elementa na 0. in 1. mestu sta koordinati, element na 2. mestu pa je vrednost). V vrstici 5 se delovanje razvrsti tako, da se obdeluje vektorje dolžine 4 po koordinati **x** in se jih paralelizira po koordinati **y**.

```
1   Func dilate;
2
3   dilate(x, y) = argmax(r, limit(x + r.x, y + r.y))[2];
4
5   dilate.vectorize(x, 4).parallel(y);
```

Slika 5.1.2.1: Koda za izvajanje širitve (del, ki se razlikuje od kode na sliki 5.1.1.1)

5.1.3 Odpiranje

Halide – prevajanje:

V implementacji, prikazani na sliki 5.1.3.1, se v vrstici 1 se deklarira objekta tipa *Func* **erosion** in **dilation**, ki se jima dodeli delovanje v vrsticah 3 in 4. Glede na to, da je za erozijo treba imeti rezultate širitve, se določi, da se širitev izvede v celoti, preden se začne izvajati erozija (vrstica 6). Nato se v vrstici 7 razvrsti izvajanje erozije z uporabo vektorizacije in paraleliziranja.

```
1   Func erosion, dilation;
2
3   dilation(x, y) = argmax(limit(x + r.x, y + r.y))[2];
4   erosion(x, y) = argmin(dilation(x + r.x, y + r.y))[2];
5
6   dilation.compute_root();
7   erosion.vectorize(x, 4).parallel(y);
```

Slika 5.1.3.1: Koda za izvajanje odpiranja

C++

V implementaciji, prikazani na sliki 5.1.3.2, se v prvih dveh vrsticah se ustvarita objekta **input**, v katerega se naloži slika, in **output**. Nato se v vrstici 3 ustvari strukturni element v obliki matrike (*MORPH_RECT*) z dimenzijami 3 × 3 (*Size(3,3)*). Prav tako se določi slikovni element znotraj strukturnega elementa, ki se ga uredi (običajno je to centralni element, tako je tudi v tem primeru (*Point(1,1)*)). V vrstici 4 se kliče funkcija *morphologyEx*, ki na podlagi tretjega argumenta izvede določeno morfološko operacijo (v tem primeru se izvede odpiranje (*MORPH_OPEN*)).

```
1 Mat input = imread("pot\do\slike");
2 Mat output;
3 Mat element = getStructuringElement(MORPH_RECT, Size(3, 3), Point(1, 1));
4 morphologyEx(input, output, MORPH_OPEN, element);
```

Slika 5.1.3.2: Koda za izvajanje odpiranja z uporabo funkcije OpenCV

5.1.4 Morfološki gradient

Halide – prevajanje:

Na začetku implementacije, prikazane na sliki 5.1.4.1, (vrstici 1 in 2) se deklarira vse objekte, ki se jih bo uporabljalo pri izvajanju operacije in razvrščanju. Nato se objektu **d** dodeli izvajanje širitve, objektu **e** erozije in objektu **g** gradienta. V primeru je objekt **r** redukcijska domena, ki predstavlja strukturni element, **limit** pa določa, kakšna vrednost se uporablja v primeru, če se

strukturni element ne nahaja popolnoma znotraj slike (v tem primeru uporabi mejne vrednosti slike).

Nato sledi razvrščanje, pri katerem se določi, da se širitev in erozija izvedeta pred gradientom. Prav tako se razdeli podatke na vektorje dolžine 4 po koordinati **x**, kar pomeni, da se bo operacija izvajala nad štirimi slikovnimi elementi hkrati na cikel. Nato se izvajanje paralelizira s funkcijo *parallel.* Na koncu se prav tako razvrsti izvajanje gradienta. Pri tem se celotno sliko razdeli na matrike velikosti 64 × 64 in paralelizira.

```
1  Var x, y, x_outer, x_inner, y_outer, y_inner, tile_index;
2  Func d, e, g;
3
4  d(x, y) = argmax(r, limit(x + r.x, y + r.y), "dilation")[2];
5  e(x, y) = argmin(r, limit(x + r.x, y + r.y), "erosion")[2];
6  g(x, y) = d(x, y) - e(x, y);
7
8  d.store_root().compute_root(); e.store_root().compute_root();
9  d.vectorize(x, 4).parallel(y);
10 e.vectorize(x, 4).parallel(y);
11
12 g.tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64).fuse(x_outer,
y_outer, tile_index).parallel(tile_index);
```

Slika 5.1.4.1: Koda za izdelovanje funkcije za izvajanje gradienta

C++

V primeru na sliki 5.1.4.2 se uporabljata funkciji za širitev in erozijo, ki sta opisani v prejšnjih primerih, tako da sta v 2. ter 3. vrstici že shranjena rezultata iz teh operacij. Nato je v 4. vrstici definirana razna matrika z enakimi lastnostmi kot vhodna slika (torej enake dimenzije in enaki tipi podatkov).

Nato sledita zanki (vrstici 6 in 7), ki predstavljata koordinate slikovnega elementa, nad katerim se izvaja morfološki gradient. Sledi odštevanje, pri katerem se tudi preverja, da razlika ne vrne negativnega števila, saj bi v tem primeru program javil napako.

Na koncu program vrne objekt tipa *Mat,* ki vsebuje spremenjeno sliko.

```
1  Mat gradient(Mat input) {
2   Mat d = dilation(input);
3   Mat e = erosion(input);
4   Mat output(input.rows, input.cols, input.type());
5
6   for (int y = 0; y < d.rows; y++) {
7    for (int x = 0; x < d.cols; x++) {
8     if (d.at<uint8_t>(Point(x, y)) - e.at<uint8_t>(Point(x, y)) >= 0) {
9      output.at<uint8_t>(Point(x, y)) = d.at<uint8_t>(Point(x, y)) -
e.at<uint8_t>(Point(x, y));
10    }
11    else {
12     output.at<uint8_t>(Point(x, y)) = 0;
13    }
14   }
15  }
16  return output;
17 }
```

Slika 5.1.4.2: Koda za izvajanje gradienta v jeziku C++

5.2 Ujemanje vzorca

Na slikah 5.2.1 in 5.2.2 je prikazana implementacija v jeziku Halide ter njen zagon. Na slikah 5.2.3 in 5.2.4 je prikazana implementacija v jeziku C++. Prva uporablja funkcijo iz knjižnice OpenCV, druga pa uporablja samo objekte iz knjižnice.

Halide – prevajanje

Koda se začne z deklaracijo vseh potrebnih objektov (vrstice 2–6). V vrstici 2 sta deklarirana objekta **x** in **y**, ki bosta določala koordinate slike. V vrstici 3 so deklarirani objekti v primeru, da se bo izvajanje razvrstilo z uporabo tilinga. Vrstici 4 in 5 vsebujeta deklaraciji argumentov, ki določata, da morajo biti slike, ki bodo uporabljene v izvajanju, črno-bele (to določa drugi argument, ki vsebuje število dimenzij matrike). Prav tako mora biti tip podatkov slike *float*. Objekt **s** bo vseboval vhodno sliko, objekt **t** pa vzorec predmeta, ki ga iščemo. V vrstici 6 sta deklarirana objekta **limit** in **score**. V vrstici 8 poskrbimo, da se v primeru, da niso vsi slikovni elementi znotraj vhodne slike, za te uporabi vrednost 1.0f. V vrstici 10 definiramo objekt **matchDom**, ki vsebuje domeno izvajanja. V tem primeru omeji izvajanje na dimenzije vzorca. V vrstici 11 je definirana še domena **searchDom**, ki omejuje delovanje funkcije. Nato se v vrstici 13 definira izvajanje primerjanja slike in vzorca. V primeru se uporablja enačba (4.2.1), ki se jo na koncu deli z zmnožkom dimenzij vzorca za pridobitev povprečij. V vrstici 15 se pridobi podatke računalnika, potrebne za prevajanje kode.

Vrstice 16–27 vsebujejo 3 različne načine razvrščanja. Kateri način se izbere, je odvisno od argumenta **i**. Prvo razvrščanje ne vsebuje nobenega razvrščanja in samo prevede kodo v

datoteko z imenom **sqdiff_unoptimized**, drugo razvrščanje združi delovanje funkcij v matrike dimenzij 64 × 64, ki se jih nato paralelizira in na koncu prevede vse skupaj v datoteko z imenom **sqdiff_tiling**. Zadnje razvrščanje združi slikovne elemente v vektorje z dolžino 16 po koordinati **x** ter paralelizira delovanje po koordinati y in prevede v datoteko **sqdiff_vectorize**.

```
1  void createAOT(int i){
2   Var x, y;
3   Var x_outer, y_outer, x_inner, y_inner, tile_index;
4   ImageParam s (type_of<float>(), 2);
5   ImageParam t(type_of<float>(), 2);
6   Func  limit, score;
7
8   limit = BoundaryConditions::constant_exterior(s, 1.0f);
9
10  RDom matchDom(0, t.width(), 0, t.height());
11  RDom searchDom(0, s.width() - t.width(), 0, s.height() - t.height());
12
13  score(x, y) = sum(matchDom, pow(t(matchDom.x, matchDom.y) - limit(x +
matchDom.x, y + matchDom.y), 2)) / (t.width() * t.height());
14
15  Target tr = get_host_target();
16  switch (i) {
17   case 0:
18    score.compile_to_static_library("sqdiff_unoptimized",{s,t},tr);
19   break;
20   case 1:
21    score.tile(x,y,x_outer,y_outer,x_inner,y_inner, 64, 64).fuse(x_outer,
y_outer, tile_index).vectorize(x_inner,
4).parallel(tile_index).compute_root();
22    score.compile_to_static_library("sqdiff_tiling",{s,t},tr);
23   break;
24   case 2:
25    score.vectorize(x, 16).parallel(y).compute_root();
26    score.compile_to_static_library("sqdiff_vectorize",{s,t},tr);
27   }
28 }
```

Slika 5.2.1: Koda za prevajanje funkcije za ujemanje vzorca

Halide – zagon

```
1  void run() {
2   std::string srcPath = "pot\do\slike";
3   std::string templPath = "pot\do\vzorca";
4
5   Image<float> s = load_image(srcPath);
6   Image<float> t = load_image(templPath);
7   Image<float> res(s.width() - t.width(), s.height() - t.height());
8
9   sqdiff_unoptimized(s.raw_buffer(), t.raw_buffer(), res.raw_buffer());
10  sqdiff_tiling(s.raw_buffer(),t.raw_buffer(),res.raw_buffer());
11  sqdiff_vectorize(s.raw_buffer(),t.raw_buffer(),res.raw_buffer());
12 }
```

Slika 5.2.2: Koda za izvajanje funkcije za ujemanje vzorca

Funkcija OpenCV

V vrsticah 3 in 4 v primeru, prikazanem na sliki 5.2.3, se tako vhodna slika kot vzorec naložita v objekte **img** (vhodna slika) in **templ** (vzorec). Nato se ustvari objekt *Mat* **result**, v katerega se bodo shranili rezultati izračunov (vrstice 6–9). Potem se izvede ujemanje vzorca s klicem funkcije *matchTemplate*. Kot argumente je treba podati vhodno sliko, vzorec, objekt, v katerega se shrani rezultat, in kateri način se uporabi (v primeru je uporabljen **SQDIFF**).

V primeru, da bi želeli pridobiti koordinate najboljšega ujemanja, se uporabi preostanek kode. V vrstici 12 se rezultati normalizirajo (glede na to, da se uporabi način **SQDIFF** in ne **SQDIFF_NORMED**, saj je ta že normaliziran). Nato se deklarirajo objekti tipa *double* **minVal** in **maxVal** ter objekti tipa *Point* **matchLoc**, **minLoc** in **maxLoc**.

V vrstici 17 se kliče funkcija *minMaxLoc*, ki pridobi najmanjšo in največjo vrednost (v primeru se shranita v objekta **minVal** in **maxVal**) ter njuni točki (v primeru se shranita v objekta **minLoc** in **maxLoc**). V vrsticah 19–20 se, odvisno od izbrane metode, pridobi vrednost najboljšega ujemanja. V primeru, da se uporablja ali metoda **SQDIFF** ali **SQDIFF_NORMED**, se uporabi najmanjša vrednost (v tem primeru vrednost iz **minVal**) in njegova lokacija (v tem primeru se uporabi objekt **minLoc**).

```
1   void MatchingMethod()
2   {
3    Mat img = imread("pot\do\slike");
4    Mat templ = imread("pot\do\vzorca");
5
6    int result_cols =  img.cols - templ.cols + 1;
7    int result_rows = img.rows - templ.rows + 1;
8
9    result.create( result_rows, result_cols, CV_32FC1 );
10
11   matchTemplate( img, templ, result, match_method );
12   normalize( result, result, 0, 1, NORM_MINMAX, -1, Mat() );
13
14   double minVal; double maxVal;
15   Point matchLoc, minLoc, maxLoc;
16
17   minMaxLoc( result, &minVal, &maxVal, &minLoc, &maxLoc, Mat() );
18  if(match_method == CV_TM_SQDIFF || match_method ==CV_TM_SQDIFF_NORMED)
{ matchLoc = minLoc; }
19  else { matchLoc = maxLoc; }
20  }
```

Slika 5.2.3: Koda za izvajanje ujemanja vzorca v jeziku C++ z uporabo funkcije OpenCV (predelana različica iz [4])

C++

Operacija, prikazana na sliki 5.2.4, se nahaja znotraj funkcije **templateMatch**, ki potrebuje za delovanje izvirno sliko (**source**) in vzorec (**templ**). V vrsticah 2 in 3 se definirata dva objekta. Prvi je **tempResult**, v katerem shranjujemo delne rezultate, drugi pa je **currROI**, v katerega shranimo del izvirne slike, ki ga primerjamo z vzorcem. V vrsticah 4–8 so podane zanke. Prvi dve (vrstici 4 in 5) določata koordinate zgornjega levega kota trenutnega dela izvirne slike, drugi dve (vrstici 6 in 7) pa sta uporabljeni pri primerjanju dela slike in vzorca. V vrstici 9 je implementirana enačba SQDIFF (4.2.1). V vrstici 12 se rezultat normalizira tako, da se ga deli z zmnožkom vzorčeve širine in dolžine ter se ga shrani v objekt **result**, ki vsebuje rezultate vseh primerjanj. V vrstici 13 se objekt **tempResult** ponastavi.

```
1  void templateMatch(const Mat source, const Mat templ, Mat &result) {
2     float tempResult = 0;
3     Mat currROI;
4     for (int i = 0; i < source.cols - templ.cols; i++) {
5        for (int j = 0; j < source.rows - templ.rows; j++) {
6           source(Rect(i, j, templ.cols, templ.rows)).copyTo(currROI);
7           for (int x = 0; x < templ.cols; x++) {
8              for (int y = 0; y < templ.rows; y++) {
9    tempResult += pow(templ.at<float>(y, x) - currROI.at<float>(y, x), 2);
10             }
11          }
12          result.at<float>(j, i) = tempResult / (templ.cols * templ.rows);
13          tempResult = 0.0f;
14       }
15    }
16 }
```

Slika 5.2.4: Koda za izvajanje ujemanja vzorca v jeziku C++ brez uporabe funkcije OpenCV

5.3 Paralelizem

Velika prednost jezika Halide je ta, da ima v primerjavi z OpenCV že možnost uporabe paralelizma pri izvajanju funkcij. To je možno doseči s klicem funkcije *parallel*, ki sprejme spremenljivko, ki služi kot indeks za podprobleme.

OpenCV sicer nima te možnost, vendar pa je mogoče uporabiti knjižnice, kot so npr. OpenMP [13] ali TBB [14]. Sami smo, za pohitritev pridobivanja vseh meritev, uporabili knjižnico OpenMP 2.0, saj je na voljo skupaj z Visual Studiom. Tega je treba omogočiti znotraj nastavitev programa.

Poglavje 6 Rezultati

6.1 Morfološki operatorji

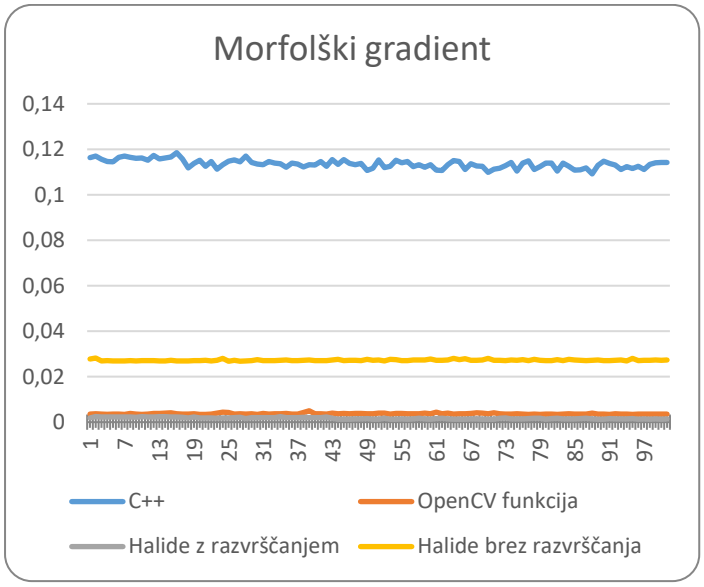
Merjenje je bilo izvedeno z osnovnim strukturnim elementom z dimenzijama 3 × 3. Izbranih je bilo več slik z različnimi dimenzijami, nad katerimi je bilo izvedenih 100 ponovitev operacij. Število vrstic implementacij in meritve časov izvajanja operacij na sliki z dimenzijami 500 × 500, 1000 × 1000 in 2000 × 2000 so prikazane v tabelah 6.1.1, 6.1.2 in 6.1.3. Graf na sliki 6.1.1 prikazuje primer izmerjenih časov.

Rezultati merjenja so dokaj podobni pri optimiziranih različicah, pri neoptimiziranih pa so razlike dokaj velike. Vse meritve kažejo na to, da je uporaba funkcije OpenCV *morphologyEx* še vedno najhitrejši način opravljanja morfoloških operacij. Ta je kar 112-krat hitrejša od različice, napisane v C++, 19-krat hitrejša od neoptimizirane različice v jeziku Halide in 4-krat hitrejša od optimizirane različice. Takoj zatem sledi različica v Halide, ki uporablja razvrščanje in je posledično optimizirana. Ta je 5-krat hitrejša od neoptimizirane različice in 32-krat hitrejša od različice v C++.

Glede ostalih dveh različic (Halide brez razvrščanja ter C++) pa se rezultata zelo razlikujeta glede na implementirano operacijo. Pri eroziji in gradientu je različica v C++ najbolj počasna, različica Halide brez razvrščanja pa približno 14-krat hitrejša od te, medtem ko je pri odprtju ravno obratno. To je zato, ker je različica v C++ narejena tako, da najprej izvede celotno erozijo in šele nato širitev, medtem ko se pri različici v Halide pri vsakem slikovnem elementu izvedeta obe operaciji hkrati, kar pomeni, da mora pri vsakem slikovnem elementu širitev počakati na rezultat erozije. Prav tako se pri tej različici vmesni rezultati ne shranjujejo in je zaradi tega izvedenih veliko redundantnih operacij.

Pri eroziji je tako, da celotna izvedba že sama po sebi sestoji iz samo enega koraka, pri gradientu pa se pri obeh različicah najprej izvedeta širitev in erozija ter šele nato odštevanje.

Prav tako je iz rezultatov razvidno, da je najkrajša koda običajno tista, ki uporablja funkcijo, implementirano v OpenCV. Sicer pa so različice v Halide vedno krajše od različic v C++. Največja razlika v dolžini kode je pri implementaciji odpiranja. Pri tem je implementacija C++ kar 11-krat daljša od implementacije v jeziku Halide brez uporabe razvrščanja, ki je celo krajša od implementacije, ki uporablja funkcijo v OpenCV.



Slika 6.1.1: Primer izmerjenih časov

Erozija	**Štev. vrstic**	**Povprečni čas izvajanja (500 × 500)**	**Povprečni čas izvajanja (1000 × 1000)**	**Povprečni čas izvajanja (2000 × 2000)**
C++	24	22,51 ms	56,6 ms	243 ms
OpenCV	6	0,2 ms	0,9 ms	3,2 ms
Halide (brez razvrščanja)	8	3,8 ms	1,7 ms	55,7 ms
Halide (z razvrščanjem)	11	0,7 ms	14 ms	5,2 ms

Tabela 6.1.1: Rezultati izvajanja erozije

Odpiranje	Štev. vrstic	Povprečni čas izvajanja (500 × 500)	Povprečni čas izvajanja (1000 × 1000)	Povprečni čas izvajanja (2000 × 2000
C++	56	29 ms	106,9 ms	457,7 ms
OpenCV	6	0,8 ms	3,6 ms	14,9 ms
Halide (brez razvrščanja)	5	520,7 ms	2000 ms	8000 ms
Halide (z razvrščanjem)	8	3,2 ms	20,5 ms	79,7 ms

Tabela 6.1.2: Rezultati izvajanja odpiranja

Morfološki gradient	Štev. vrstic	Povprečni čas izvajanja (500 × 500)	Povprečni čas izvajanja (1000 × 1000)	Povprečni čas izvajanja (2000 × 2000)
C++	71	29,4 ms	113,6 ms	521,6 ms
OpenCV	6	0,8 ms	3,7 ms	14,7 ms
Halide (brez razvrščanja)	11	5,5 ms	27,2 ms	117,6 ms
Halide (z razvrščanjem)	12	0,6 ms	1,7 ms	8,9 ms

Tabela 6.1.3: Rezultati izvajanja morfološkega gradienta

6.2 Ujemanje vzorca

Za testiranje je bila izbrana slika z dimenzijama 500 × 360, vzorec pa z dimenzijama 33 × 48. Tabela 6.2.1 prikazuje povprečni čas izvajanja operacije v Halide z in brez uporabe razvrščanja ter z in brez uporabe funkcije OpenCV *matchTemplate* v jeziku C++ ter dolžino kode. Iz teh časov je očitno, da je funkcija iz OpenCV najhitrejša. Ta je 7-krat hitrejša od različice, napisane

v jeziku Halide z uporabo razvrščanja, in kar 474-krat hitrejša od različice, napisane v jeziku C++. Prav tako je iz izmerjenih časov razvidno, kako lahko razvrščanje vpliva na čas izvajanja, saj je različica z razvrščanjem 33-krat hitrejša od različice brez razvrščanja.

Ujemanje vzorca	**Štev. vrstic**	**Povprečni čas izvajanja**
C++	17	14226,98 ms
OpenCV	4	31,886 ms
Halide (brez razvrščanja)	12	7699,712 ms
Halide (z razvrščanjem)	13	231,619 ms

Tabela 6.2.1: Rezultati izvajanja ujemanja vzorca

Poglavje 7 Sklepne ugotovitve

V diplomskem delu sta opisana programski jezik Halide in knjižnica OpenCV. Poleg tega so prikazani tudi primeri kode, ki vsebujejo implementacije morfoloških operacij in ujemanje vzorca. Glede na to, da je Halide funkcijski jezik, je iz primerov razvidno, kako zelo so si kode različne. Še posebej je ta razlika očitna pri različici C++, saj vsebuje zanke, ki določajo potek operacije, zato lahko posledično podvojijo dolžino kode in zmanjšajo njeno preglednost.

Pomembno je, da je v primeru, da želimo sami implementirati operacije (torej ne želimo uporabljati že implementirane funkcije v knjižnici OpenCV) v jeziku C++, Halide boljša alternativa, saj omogoča enostavno implementacijo operacij in je v večini primerov bistveno hitrejši, s tem pa bolj ekonomičen in racionalen. Zaradi delitve kode na implementacijo in optimizacijo operacij je mogoče zelo hitro testirati različna razvrščanja in s tem najti najbolj optimalno.

Pred kratkim so avtorji jezika objavili prvo različico avtomatskega iskanja optimalnega razvrščanja in objavili članek [7]. Prav tako so že objavljene njegove različice.

V drugem delu so prikazani izvajalni časi vseh implementacij in dolžina kode. Pri tem so bili rezultati v prid uporabi funkcij, implementiranih v knjižnici OpenCV. Sicer pa je bil Halide vseeno zelo hiter v primerjavi z implementacijami operacij v C++ (razen v implementaciji odprtja), kar kaže na to, da je jezik uporaben. Prav tako je iz rezultatov razvidno, kako zelo lahko dodajanje razvrščanja vpliva na izvajalni čas funkcij. Glede na to, da je ta programski jezik relativno mlad (nastal je leta 2012), ima veliko potenciala za izboljšanje.

Literatura

[1]. G. Agam. Introduction to programming with OpenCV, Department of Computer Science, 2006 [Online]. Dosegljivo: http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html [Dostopano 1.11.2016].

[2]. J. F. Rivest, P. Soille, S. Beucher. Morphological gradients, *J. Electron. Imaging,* zv. 2, št. 4, str. 326-336, 1993.

[3]. R. C. Gonzalez, R.E. Woods. Digital Image Processing Third Edition, str. 653 – 658, Prentice - Hall, Inc. Upper Saddle River, NJ, USA 2006.

[4].Template Matching – OpenCV 2.4.13.1 documentation [Online]. Dosegljivo: http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html [Dostopano 6.7.2016].

[5]. J. Ragan-Kelley, C. Barnes, A. Adams, S. Amarasinghe, F. Durand, S. Paris. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, *ACM SIGPLAN Notices - PLDI '13*, zv. 48, št. 6, 2013.

[6]. J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, F. Durand. Decoupling Algorithms from Schedules Easy Optimization of Image Processing Pipelines, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2012,* zv. 31, št. 4, 2012.

[7]. R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, K. Fatahalian. Automatically Scheduling Halide Image Processing Pipelines, *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH,* zv. 35, št. 4, str. 519-530, 2016.

[8]. Halide documentation [Online]. Dosegljivo: http://halide-lang.org/docs/ [Dostopano 2.11.2016].

[9]. OpenCV spletna stran [Online]. Dosegljivo: http://opencv.org/ [Dostopano 5.10.2016].

[10]. OpenCV Documentation [Online]. Dosegljivo: http://opencv.org/documentation.html [Dostopano 5.10.2016].

[11]. OpenCV Download Statistics [Online]. Dosegljivo: https://sourceforge.net/projects/opencvlibrary/files/stats/timeline?dates=2001-09-20+to+2016-11-28 [Dostopano 28.11.2016].

[12]. Slika, uporabljena za prikaz delovanja ujemanja vzorca [Online]. Dosegljivo: https://commons.wikimedia.org/wiki/File:Observation_balloon_RAE-O982a.jpg [Dostopano 15.11.2016].

[13]. OpenMP homepage [Online]. Dosegljivo: http://www.openmp.org/ [Dostopano 7.12.2016].

[14]. Threading Building Blocks homepage [Online]. Dosegljivo: https://www.threadingbuildingblocks.org/ [Dostopano 7.12.2016].